Tool for Translating Simulink Models into Input Language of a Model Checker

Meenakshi B., Abhishek Bhatnagar, and Sudeepa Roy*

Honeywell Technology Solutions Lab
Bangalore 560076, India
Meenakshi.Balasubramanian@honeywell.com

Abstract. Model Based Development (MBD) using Mathworks tools like Simulink, Stateflow etc. is being pursued in Honeywell for the development of safety critical avionics software. Formal verification techniques are well-known to identify design errors of safety critical systems reducing development cost and time. As of now, formal verification of Simulink design models is being carried out manually resulting in excessive time consumption during the design phase. We present a tool that automatically translates certain Simulink models into input language of a suitable model checker. Formal verification of safety critical avionics components becomes faster and less error prone with this tool. Support is also provided for reverse translation of traces violating requirements (as given by the model checker) into Simulink notation for playback.

1 Introduction

Model Based Development (MBD) is a concept of software development in which *models* are developed as work products at every stage of the development life-cycle. Models are concise and understandable abstractions that capture critical decisions pertaining to a development task and have semantics derived from the concepts and theories of a particular domain. Models supersede text and code as the primary work products in MBD and most development activities are carried out by processing models with as much automation as possible.

MBD is known to improve the quality of the product being developed. Formal models of design are used for proving the design correct with respect to functional requirements, identifying errors early in the life-cycle. Automatic methods for generating code and test cases helps to reduce coding errors and save total development time spent in coding and testing phases.

Formal verification techniques like *theorem proving* and *model checking* are well-known to reduce defects in the design stage by checking if a design meets its functional requirements [9]. Presence of formal models in MBD gives room for analysis using formal verification. Both MBD and formal verification are practices that put emphasis on detecting design errors (that have high leakage rate) rather than implementation errors (that have low leakage rate).

* Presently at Google, Bangalore, India.

Z. Liu and J. He (Eds.): ICFEM 2006, LNCS 4260, pp. 606–620, 2006.

DO-178B [1] standard produced by Radio Technical Commission for Aeronautics Inc. defines guidelines for development of avionics software and is the accepted means of certifying all new avionics software. DO-178B is obsolete with respect to MBD process but recognizes formal methods as a way to prevent and eliminate requirements, design and code errors throughout the development lifecycle. The benefit of formally verifying models at design stage is also validated by its successful use in various industrial examples [9].

In spite of all the above advantages, formal verification has not been successfully integrated into many development processes. The main issues are related to making it easy to use by the system engineers. Formal verification tools typically do not support standard design modeling notations but have their own notations related to the theories of the tool. The extra effort to learn the notations to use these tools is usually not welcome due to the delays it causes in development time. Consequently, there is a need to automate the formal verification process as much as possible for use by system engineers.

One possible step towards automation is to make formal verification tools available in notations that system engineers typically use. Mathworks tools like Simulink [2], Stateflow [3] etc. are extensively used in Honeywell for avionics software development. For system engineers to formally verify their design, it would be ideal if these modeling tools can automatically link to suitable model checking tools. We meet such a need in this paper by developing a *translator* from Simulink to the model checker NuSMV [4]. NuSMV is an open source symbolic model checker jointly developed by ITC-IRST, CMU, University of Genova and University of Trento. The translator takes a Simulink model as input and generates an equivalent NuSMV model.

The translator supports all the basic blocks that constitute a *finite state* subset of Simulink, i.e., any Simulink model obtained by putting together these blocks constitutes a finite state machine. The model generated by the translator can be formally verified against temporal logic requirements using the NuSMV model checker. We are working on providing support for specifying requirements by using a template based tool along the lines of the specification pattern system developed in [11]. These two tools put together would constitute a full-fledged verification tool for Simulink models.

Some other tools have been developed for formally verifying Simulink models. Commercial tools like SCADE design verifier [8], Embedded Validator [5] support formal verification of Simulink models against safety properties and work with their customized library of Simulink blocks, again mainly blocks from the discrete library. These tools were not expressive enough to translate some of models used in Honeywell, one such model involving an avionics triplex sensor voter is presented in the paper.

Checkmate [6] is a research tool developed to translate Simulink models into hybrid automata notation and verification is done using abstraction and certain semi-decision procedures involving reachability analysis of hybrid automata which are not guaranteed to terminate. Since Checkmate can translate Simulink models into hybrid automata, the translation also supports certain continuous

basic blocks of Simulink. Thus, even though a larger set of Simulink models can be translated, fully automated verification of models is not possible as the reachability analysis procedures of the considered class of hybrid automata are not guaranteed to terminate.

Our algorithm works with standard Simulink notation and semantics and the models are translated into NuSMV which is an open source verification tool supporting the fully automatic technique of model checking. This achieves the main goal of providing a fully automated formal verification support to system engineers using MBD based on Simulink models.

2 Preliminaries

We briefly describe Simulink and NuSMV tools in this section.

2.1 Simulink

Simulink is a computer aided design tool widely used in the aerospace industry to design, simulate and auto code software for avionics equipment [2]. A Simulink model of a system is a hierarchical representation of the design of the system using a set of blocks that are interconnected by lines. Each block represents an elementary dynamic system that produces an output either continuously (continuous block) or at specific points in time (discrete block). The lines represent connections of block inputs to block outputs. Simulink provides various libraries of such blocks and in addition, some additional blocks can also be user-defined. Interconnected blocks are used to build sub-systems which in turn are put together to form a system model.

Simulink, considered as a de-facto standard in control design, is proven to be expressive enough to model many avionics systems and offers extensive simulation capabilities for de-bugging the design model.

2.2 NuSMV Model Checker

NuSMV [4] is a symbolic model checker based on Binary Decision Diagrams (BDDs) [7]. It allows for the description of systems as finite state machines, both synchronous and asynchronous. Specifications regarding the system can be given as Computation Tree Logic (CTL) and Linear Temporal Logic (LTL) formulas. Model checking algorithms in NuSMV check if the system meets the specifications using BDD-based and SAT-based model checking techniques and are ideally suited for verifying hardware designs.

The data flow block diagram of a Simulink model resembles control flow like that of hardware design even though Simulink models are finally implemented in software. This is the main reason behind choosing NuSMV as the target model checking tool for formally verifying Simulink diagrams apart from the fact that NuSMV is an open source tool. Also, NuSMV being a symbolic model checker is capable of handling systems with huge state space size. We illustrate this fact by

applying the translator algorithm on the Simulink model of an avionics triplex sensor voter. The details are described in a subsequent section.

NuSMV input language. The input language of NuSMV is designed to allow for specification of system models as finite state machines. The data types provided by the language are Booleans, bounded integer sub-ranges, symbolic enumerated types and bounded arrays of these basic data types.

Complex system models can be described by decomposing it into *modules*. Each module defines a finite state machine and can be instantiated many times. Modules can be composed either synchronously or asynchronously to get the full system description. In synchronous computation, a single step in the composed model corresponds to a single step in each of the modules. In asynchronous computation, a single step in the composed model corresponds to a single step performed by exactly one module.

3 The Translator Algorithm

We describe the translator algorithm from Simulink models into NuSMV model checker in this section along with details about the execution semantics and the reverse translation. Working of the algorithm along with its use in formal verification of Simulink models will be illustrated in the next section with an example from the avionics domain.

3.1 Description of the Algorithm

The translator algorithm takes the MDL file format (textual representation) of the Simulink model as input and outputs its equivalent model in the input notation of NuSMV as described in Section 2.2.

Each basic block in Simulink (in the libraries supported by the translator algorithm) is translated into its equivalent module in NuSMV. For a given Simulink model, the NuSMV model that is output by the translator varies with the type of input ports of the Simulink model. Basic blocks of Simulink are generic, for example, the basic block corresponding to addition can add two scalars or two vector inputs, type matching and conversion are taken care of automatically. However, this is not the case with NuSMV, the module that adds two scalar inputs is different from the one that adds two vector inputs. Consequently, there is one NuSMV module corresponding to a given basic block and input type in Simulink.

A *library of routines* to generate NuSMV modules equivalent to basic blocks in Simulink are written to be re-used while generating NuSMV models from given Simulink models. The routines in this library respect the correspondence between basic blocks and modules mentioned above. For example, consider the standard relational operator block in Simulink given in Figure 1. Assume that the first input (`in1`) to the block is a vector of length 2, the second input (`in2`) is

a scalar and the operation being checked for is $\leq$. The NuSMV module equivalent to the relational operator basic block is given below:

```
MODULE _relational_operator_2(in1, in2)

VAR
    out : array 0..1 of boolean;

ASSIGN
    out[0] := in1[0] <= in2;
    out[1] := in1[1] <= in2;
```

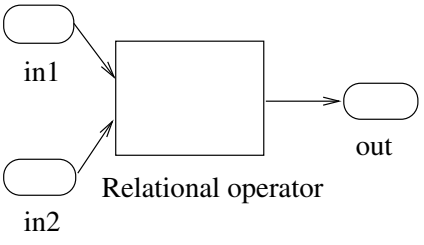

Fig. 1. Relational operator block in Simulink

The above module will be generated by a routine in the library to be re-used whenever the relational operator block with two inputs (of types as above) is being used in a model.

The translator algorithm is divided into the following steps:

1. **Parsing the model:** The model is read from its textual representation, irrelevant information involving the graphics of the model (like color, font size etc.) are discarded and information regarding blocks and subsystems, input and output ports, variables, inter-connection of blocks etc. is extracted.
2. **Computing input type of blocks and sub-systems:** In this step, a walk through the output of the graph structure extracted from step (1) is done wherein the type of input of each block is computed depending upon the output of preceding blocks.
   (a) For each block of source library of Simulink, input types of all the connected blocks is populated. Output type information for source block can be calculated directly from Simulink model.
   (b) If depending upon the input type, any decision regarding block output type can be taken (For example, in the case of Add block, if one of the inputs is a vector of n-dimension then output will be of n-dimension, where $n > 1$, or when all input port types are of 1-dimension then output will also be of type 1-dimension), then all the blocks further connected to this block are populated with input port type.

(c) The above step is continued for all the blocks in the graph until a block for which output type cannot be computed is reached. At this stage, control is transferred to the parent of this block in the model and the previous step and this one are repeated for the other connected blocks from the output port of the parent. This is done iteratively till all the blocks connected to one of the source blocks (in the first step above) are exhausted.

Note that this step is guaranteed to terminate as the input model has a fixed number of blocks.

If block is of type subsystem, then, blocks inside this sub-system are populated as per step 2 above. Once output port is reached, all the blocks connected to this output port of the subsystem are populated as done in step 2(c).

3. **Writing the final file:** In the final step, routines from the library described above are used to write the NuSMV model wherein each basic block is replaced by its equivalent modules(s). Here again, sub-systems are translated first respecting the hierarchy in the model.

Notice that the translation preserves the structure (hierarchy of the blocks, their names and interconnections) of the input Simulink model. The NuSMV model output by the translator follows the same hierarchical structure as the input Simulink model and variable names are also retained to be the same. Also, there is one module in NuSMV model corresponding to each basic block in the Simulink model. These features are important in MBD for answering traceability related questions and also for the verification of requirements as some of them might be specified by fully exploiting the structure in the model.

The above algorithm has been implemented and has been tested on some examples to check for the translation preserving the model. We present a detailed example involving the translation of Simulink model corresponding to an avionics triplex sensor voter in the next section.

3.2 Simple Abstraction Feature

The size of the translated NuSMV model is an important factor to make it amenable for verification. Many abstraction techniques are used to avoid the famous *state space explosion problem*. The fact that NuSMV is a symbolic model checker comes in useful here as such model checkers are well-known to handle systems with large state space size.

We have provided a simple state abstraction feature to be able to model check Simulink models that are too huge even for symbolic model checkers like NuSMV. While running the above translation algorithm, the system engineer has the option of bounding the ranges of certain/all variables that occur in the model. If no ranges are specified, the translator assumes maximum range. Otherwise, ranges of certain variables can be bounded by the system engineer and are incorporated into the translated model. This will help in reducing the state space size whenever required, while retaining the features required for verification of requirements.

3.3 Execution Semantics

Some points are worth noting regarding the translator algorithm preserving the behavior of the Simulink model. Semantics of systems modeled using Simulink is presented through simulations, which are done by sampling the data in the model. As mentioned above, Simulink models have both discrete and continuous blocks. *Sample time* parameter talks about the rate at which the states of the Simulink model are updated. The sample time is by definition, continuous for continuous blocks and is explicitly specified by the user for discrete blocks.

The scope of the translator algorithm presented in this paper is restricted to the discrete blocks of Simulink as the model checker NuSMV is capable of modeling discrete finite state systems only. We assume that one sample time in the Simulink model is equivalent to one *execution step* (modeled by a transition from one state to another) in the NuSMV model. The NuSMV model is equivalent to the given Simulink model as generated by the translator with respect to this assumption. Also, as noted in the previous section, for a given Simulink model, the NuSMV model generated by the translator varies with the type of input ports. Given the above, the translator algorithm preserves the given Simulink model as follows: at any point of execution, for every *state* of the Simulink model (given by the values which all the variables in the model take), there is a corresponding state in the NuSMV model wherein the variables take the same values. Also, transitions between states that arise because of change of values of certain variables in the Simulink model also result in corresponding transitions between corresponding states in the NuSMV model.

3.4 Finite State Simulink Models

As mentioned in the previous section, the scope of the translator is restricted to discrete Simulink models only, mainly due to the fact that the model checker NuSMV is capable of modeling discrete systems only. Here again, the model checker NuSMV is a finite state verification tool, that is, the class of models that can be formally verified using NuSMV are finite state machines. Consequently, the translator can support all the basic blocks of Simulink that, when put together, form a finite state model of a system.

Basic blocks of Simulink are organised into libraries of those with similar properties. In the translator algorithm, all the blocks of the signal routing, logic and bit operations, math operations (discrete), sources, discontinuities and discrete libraries are supported as of now with integer and Boolean data types for variables. As we illustrate in a subsequent section, the translator algorithm is expressive enough to translate non-trivial avionics Simulink models that are built using basic blocks from these libraries. Detailed list of the various blocks (listed within the libraries they belong to) are given below.

- Signal routing library
  - Demux and mux blocks
  - Switch block

  - Selector block
  - Multi-port switch, index vector blocks
  - Merge block
- Logic and bit operations library
  - Relational block
  - Logical block
  - Interval test block
  - Interval test dynamic block
  - Compare to zero, compare to constant blocks
- Math operations library
  - Sum, add, subtract and sum of elements blocks
  - Product, divide and product of elements blocks
  - Abs block
  - Unary minus block
  - Sign block
  - Bias block
  - Min-max block
  - Gain block
- Sources library
  - Ground block
  - Constant block
  - In port block
  - Uniform Random Number
  - Step
  - Counter Free Running
  - Counter Limited
  - Read From File
- Discontinuities library
  - Saturation block
  - Saturation dynamic block
  - Dead zone block
  - Dead zone dynamic block
  - Wrap to zero block
  - Coulomb and vicious function block
- Discrete blocks library
  - Unit delay and integer delay blocks
- Sinks blocks Library
  - Out port Block

3.5 Reverse Translation

NuSMV (and many other model checking tools) take a system model and a requirement as input and provide a yes/no answer depending on whether the system satisfies the requirement or not respectively. In the latter case, a system run violating the requirement is also output by the model checking tool as evidence to the fact the system does not meet the requirement. This feature is very useful in de-bugging the system design to meet the requirement.

In order to facilitate the Simulink system engineer to de-bug the model, we provide a *reverse translation routine* that takes a system run produced by NuSMV (as counter example) as input and translates it back into a textual notation that a Simulink designer can understand.

Since the translation algorithm preserves the structure of the input model, the counter example output by NuSMV reveals the structure fully in its description. Consequently, the reverse translation routine is a simple scripting program that re-writes the example in a notation that a Simulink designer can understand and simulate. Simulation of a violating run helps in de-bugging the design.

4 Sensor Voter Example

We describe an example involving a Simulink model used in digital flight control, namely that of an *avionics triples sensor voter*. This model was automatically translated into NuSMV by the algorithm and various computational and fault-handling requirements of the model were verified using NuSMV.

4.1 Triplex Sensor Voter

Almost all digital flight control systems utilize redundant hardware to meet high reliability requirements. Use of redundant hardware poses two problems: distinguishing between operational and failed units and computing the "mean" value of the various units for use by other components. A key part of redundant systems are redundant sensors and algorithms that focus on managing redundant sensors to provide a high integrity measurement for use by down-stream control calculations. We consider a *voter algorithm* that manages three redundant sensors in this paper. This class of algorithms is applicable to a variety of sensors used in modern avionics, including air data sensors, surface position sensors etc. The voter model has been translated by hand into the model checking tool SMV and many requirements were verified [10]. We now describe the sensor voter model and our work related to its formal verification using the translation algorithm.

Sensor voter operation. Simulink model corresponding to the sensor voter is described in Figure 2. The voter takes input from three sensors and produces a single reliable sensor output. Each sensor produces a measured data value and a self-check bit indicating whether or not the sensor considers itself to be operational.

The operation of the voter algorithm is described in the steps below:

- All valid sensor data are combined to produce output.
- If three sensors are available, a weighted average is used in which an outlying sensor value is given less weight than those that are in closer agreement.
- If only two sensors are available, a simple average is used.
- If only one sensor is available, it becomes the output.

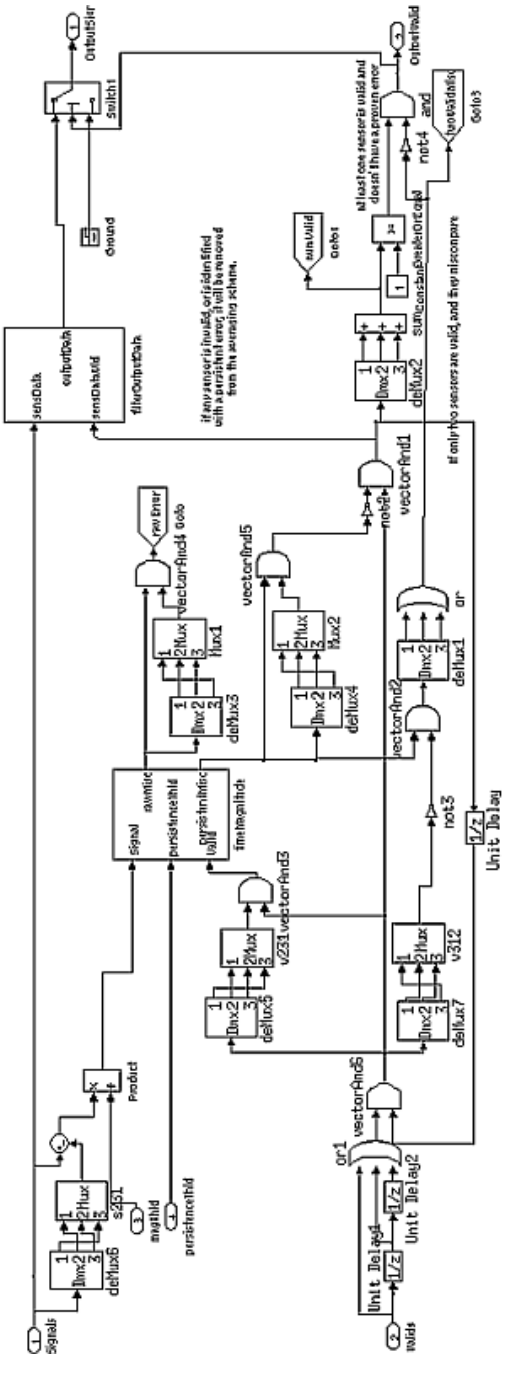

Fig. 2. Simulink model of avionics triplex sensor voter

A faulty sensor value is not used in failure comparisons or in the production of the output signal. The following are the mechanisms by which a faulty sensor can be detected and eliminated:

- Any sensor input whose own self-check bit is false is not used.
- Next, all the sensor values are compared two at a time. If difference exceeds threshold, magnitude error is set. If magnitude error persists longer than magnitude threshold, persistent miscompare is set. (threshold, magnitude error and persistent miscompare are variables in the model).
- If sensors 1 and 2 have persistent miscompare and so do sensors 2 and 3, sensor 2 is flagged as persistent sensor error and is not used.
- If only two sensors are valid and then miscompare, output depends on the self-check bit.

Requirements of the sensor voter were either relating to the value of the output signal computed by the voter or they were fault handling requirements that talk about mechanisms to detect and isolate faulty sensors. We translated the requirements manually into CTL formulas for verification using NuSMV.

Sensor voter modeling. In order to perform formal verification, it is just not sufficient to translate the sensor voter model into NuSMV. We need to model the *environment* in which the voter is used so that faults can be injected into the model externally. The environment was modeled by using a *world block* that acts as an abstraction of all the components that provide inputs for the sensors to measure. There are also three blocks corresponding to the sensors that model the physical sensors that generate the measured signal. The sensor blocks were used to inject faults to test the ability of the voter to identify them. These blocks were added to the original Simulink model to create a model amenable to formal verification.

Formal verification. The Simulink model of sensor voter (as given in Figure 2) was modified by adding the world and sensor blocks as described above. The new model constitutes what we call a *fault model* where different values can be injected to perform "what if" analysis to check if the requirements are met.

The translator tool was invoked to translate the fault model of sensor voter into NuSMV. Since the sensor voter model is big, the NuSMV code of the model as generated by the translator is not fully presented. Figure 3 gives a snapshot of part of the NuSMV model containing the declarations of the modules corresponding to the sub-systems and the blocks in the outermost level of the sensor voter model.

We now present the results of verifying two requirements related to the sensor voter. The requirements were given as CTL formulas to NuSMV.

1. The first property relates to the requirement that detection and elimination of a faulty sensor is final, i.e. once a sensor is detected and eliminated as being faulty, it is never available as an active sensor again. This requirement

```
VAR
      Constant : __constant_18();
      GreaterOrEqual : __relational_operator_4(sum._1000, Constant
      Ground : __ground_1();
      Mux1 : __subsystem_6(deMux3._3, deMux3._1, deMux3._2);
      Mux2 : __subsystem_7(deMux4._3, deMux4._1, deMux4._2);
      Product : __product_1(Sum.out1, magThld);
      Sum : __add_1(InSignals, s231.Mux);
      Switch1 : __switch_1(filterOutputData.outputData, and._1000,
      Unit_Delay : __unit_delay_4(vectorAnd1._1000);
      Unit_Delay1 : __unit_delay_5(Valids);
      Unit_Delay2 : __unit_delay_6(Unit_Delay1.out1);
      and : __subsystem_8(GreaterOrEqual.out1, not4._2);
      deMux1 : __subsystem_9(vectorAnd2._1000);
      deMux2 : __subsystem_10(vectorAnd1._1000);
      deMux3 : __subsystem_11(timeMagnitude.rawMisc);
      deMux4 : __subsystem_12(timeMagnitude.persistentMisc);
      deMux5 : __subsystem_13(vectorAnd6._1000);
      deMux6 : __subsystem_14(InSignals);
      deMux7 : __subsystem_15(vectorAnd6._1000);
      filterOutputData : __subsystem_16(InSignals, vectorAnd1._100
      not2 : __subsystem_29(vectorAnd5._1000);
      not3 : __subsystem_30(v312.Mux);
      not4 : __subsystem_31(or._1000);
      or : __subsystem_32(deMux1._1, deMux1._2, deMux1._3);
      or1 : __subsystem_33(Valids, Unit_Delay1.out1, Unit_Delay2.o
      s231 : __subsystem_34(deMux6._2, deMux6._3, deMux6._1);
      sum : __subsystem_35(deMux2._1, deMux2._2, deMux2._3);
      timeMagnitude : __subsystem_36(Product.out1, persistenceThld
      v231 : __subsystem_40(deMux5._2, deMux5._3, deMux5._1);
      v312 : __subsystem_41(deMux7._3, deMux7._1, deMux7._2);
      vectorAnd1 : __subsystem_42(not2._2, vectorAnd6._1000);
      vectorAnd2 : __subsystem_43(timeMagnitude.persistentMisc, no
      vectorAnd3 : __subsystem_44(v231.Mux, vectorAnd6._1000);
      vectorAnd4 : __subsystem_45(timeMagnitude.rawMisc, Mux1.Mux)
      vectorAnd5 : __subsystem_46(timeMagnitude.persistentMisc, Mu
      vectorAnd6 : __subsystem_47(or1._1000, Unit_Delay.out1);
DEFINE
      OutputSignal := Switch1.out1;
      OutputValid := and._1000;
      __Goto__1[0] := vectorAnd4._1000[0];
      __Goto__1[1] := vectorAnd4._1000[1];
      __Goto__1[2] := vectorAnd4._1000[2];
      __Goto1__1 := sum._1000;
      __Goto3__1 := or._1000;
      __Goto5__2[0] := timeMagnitude.__Goto5__1[0];
      __Goto5__2[1] := timeMagnitude.__Goto5__1[1];
      __Goto5__2[2] := timeMagnitude.__Goto5__1[2];
```

Fig. 3. Main module of the NuSMV code of triplex sensor voter

is specified by using the following CTL formula which specifies that there is no execution path where the number of valid sensors increases.

```
 AG (
(voting3signals.OutputValid = 0 ->
!EF voting3signals.OutputValid = 1)
& (voting3signals.OutputValid = 1 ->
!EF voting3signals.OutputValid = 2)
& (voting3signals.OutputValid = 2 ->
!EF voting3signals.OutputValid = 3))
```

NuSMV reported this specification to be true. We first ran verification with unbounded ranges and since NuSMV model checker took about a week to produce the results, we tried verification algorithm by bounding the range of one variable, namely, unit delay, to vary from -30 to 30. With this option, the state space size reduced drastically and verification of the model with respect to the above property completed with the same result within a few seconds. This acts as a good illustration of the static abstraction feature explained earlier.

2. The second requirement relates to fault handling requirements of the sensor voter. If the number of valid sensors is 2 and the voter output is valid and the second sensor becomes faulty then in the future, the number of valid sensors is 1 and the voter output is still valid.

```
 AG (
((voting3signals._Goto1_1 = 2
& voting3signals.OutputValid) & fault2) ->
AF (voting3signals._Goto1_1 = 1
& voting3signals.OutputValid))
```

NuSMV reported this property to be false and gave a violating run. The property turned out to be false due to a problem with our fault model (and not the voter model). We had modeled the sensors in such a way that a faulty sensor will never exhibit any faulty behavior in terms of the way in which the values are measured.

5 Model Based Formal Analysis

Engineers traditionally perform well-established but, informal analysis techniques like Fault Tree Analysis (FTA) and Failure Modes and Effects Analysis (FMEA) for checking for safety requirements of their system. These techniques are well established and are used extensively during the design of safety critical systems. Despite this, most of the techniques are highly subjective and dependent on the skill of the practitioner as they are based on informal system models that are derived in an ad hoc fashion. This results is excessive consumption of resources and time.

Due to these reasons, there is an increasing shift towards using MBD techniques for analysis of the design of safety-critical systems. In this approach, various development activities including design and simulation, verification, testing

and code generation are based on a formal model of the system. The presence of formal models makes the development process amenable to using formal verification techniques like model checking. However, there are certain gaps to be filled for model checking techniques to be directly used by system engineers.

We already discussed one of the gaps in the introduction, namely that of the model checking notations not being easy-to-use by system engineers. The translator algorithm presented in this paper fills this gap. Few more questions need to be answered to fully integrate techniques like model checking with MBD. The first among them is to be able to formalize a *fault model* of the system under test. Fault model captures the various ways in which the components of the system can malfunction. This information is provided by modeling the entities that the system interacts with so that faults can be externally introduced into the system without altering the system model. For example, in the verification of sensor voter presented in the previous section, fault model comprises of abstract models of the sensors and the world block gives data to the sensors. This step has to be done by the system engineers themselves, manually.

The second gap comes from the requirements side. Functional requirements which ensure safe behavior of the system are usually specified in text document along with other requirements. The safety properties must be expressed in some formal notation to support automated analysis. There are several formal specification languages like CTL, LTL, finite state machines etc. that are supported by many model checkers. We are working on automating this step by exploiting the work done in [11] where the authors provide a repository of commonly occurring specification patterns in the specification of concurrent, reactive systems. There is a mapping from these specification patterns to a number of formalisms that are supported by tools for formal analysis. LTL and CTL languages that are supported by NuSMV are also provided amongst the formalisms.

A template-based description of these specification patterns is being developed with facilities to include model specific values to the specification templates. These will be translated into equivalent CTL/LTL formulas so that they can be fed into the model checker NuSMV for verification automatically. This step would fill all the gaps that exist for fully automated use of model checking techniques by Simulink system engineers.

6 Conclusions

We have presented a translator algorithm that translates a subset of Simulink into input language of the model checker NuSMV. The subset of Simulink blocks supported by the translator is expressive enough to translate many interesting classes of avionics models like the avionics triplex sensor voter presented in this paper. The tool aids in automating formal verification of Simulink models and will be of valuable use in model based formal safety analysis of systems.

References

1. DO-178B guidelines. Available from: `http://www.rtca.org/`.
2. Simulink web page: `http://www.mathworks.com/products/simulink/`.
3. Stateflow web page: `http://www.mathworks.com/products/stateflow/`.
4. NuSMV web page: `http://nusmv.irst.itc.it/`.
5. Embedded Validator web page: `http://www.dspaceinc.com/ww/en/inc/home/products/sw/pcgs/automatic_model_validation.cfm`.
6. Checkmate web page: `http://www.ece.cmu.edu/~webk/checkmate/`.
7. Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.
8. Jean-Louis Camus and Bernard Dion. Efficient development of airborne software with scade-suite. Technical report, Esterel Technologies, 2003.
9. Edmund M. Clarke and Jeannette M. Wing. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.
10. Samar Dajani-Brown, Darren Cofer, Gary Hartman, and Steve Pratt. Formal modeling and analysis of an avionics triplex sensor voter. In *Proc.SPIN*, pages 34–48. Springer, 2003.
11. Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Property specification patterns for finite-state verification. In Mark Ardis, editor, *Proc. 2nd Workshop on Formal Methods in Software Practice (FMSP-98)*, pages 7–15, New York, 1998. ACM Press.